QuickSort

> Mattia Bergomi, Patric Müller

Contents

Contents

Contents

The Problem of Sorting

- The Problem of Sorting

  Sorting an Array of given Elements is one of the most important algorithmic Problem of the last Century.

  Such a Problem does not only have a theoretical importance, but represents a Situation that often appears in Praxis and is needed as subroutine in a lot of other algorithms (like searching, storing, etc.) and therefore needs an Optimal way to be solved.

Definition & Classification

> Definition

In computer science a **Sorting Algorithm** is defined as an algorithm that puts the elements of a given array (list) in a certain order (according to a given total order: often numerical or lexicographical). More formally, the output must satisfy the following two obvious conditions:

3. The Output is in non-decreasing order. (→ Sorted)
4. The Output is a permutation of the Input. (→ Unchanged)

> Classification

The classification of Sorting Algorithm can be classified using more criteria:

- The Computational Complexity (Running Time), expressed in "Big O" Notation.
- Memory Usage
- Stability (order of equal keys preserved)
- Whether or not they are a Comparison Sort.
- …

"Big O" Notation and Analysis Type

- "Big O" Notation (Landau Notation)

  "Big O" Notation is used to describe how the size of the input (usually "n") affects an algorithm's running Time (or Memory Usage) in the Asymptotics.

  In fact we want to describe an Expression just by the dominant Term who characterizes the behavior of the function, as n tends to infinity.

  Informally, the Symbols (and Meanings) are:

≤	f(n) = **O**(g(n))	*g(n) is an Upper Bound*
≥	f(n) = **Ω**(g(n))	*g(n) is a Lower Bound*
=	f(n) = **Θ**(g(n))	*g(n) is an optimal Bound*

"Big O" Notation and Analysis Type

- Analysis Type

  Computing the Running Time of an Algorithm can be done in many ways, but normally the number of comparisons is analyzed in the **Best-, Worst- and Average-Case**.

  Best- and Worst-case are usually given by a concrete Input Example, while the Average-Case is usually computed solving a Recursion (directly or using other tools).

  Usually one wants to compute the Average-Case and, if possible, the probability of the Worst-Case to happen (hopefully low).

Other Sorting Algorithms

- Other Algorithms

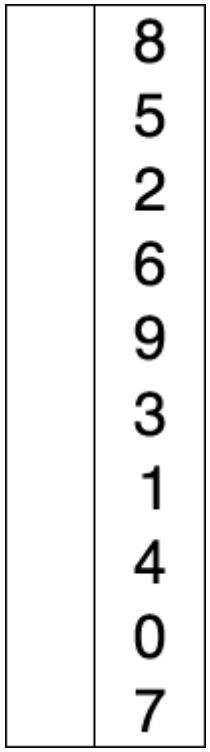
  In more than 50 years a lot of algorithms have been developed for this purpose, everyone having his strong- and weak-points, using different data structures in order to optimize the behavior of the algorithms themselves.

  Here we give just a couple of examples of them, explaining only the main Idea behind and giving the result for the Running Times.

SelectionSort *(Stable, Comparison Sort)*

Selection Sort acts just like a human would: it scans the whole array to find the smallest number, than the second-smallest, and so on…

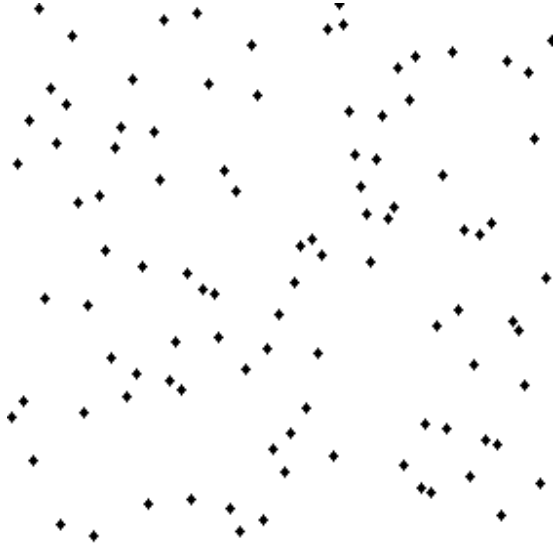
The distribution of the Input is not important, and the number of performed comparisons is always the same.



BubbleSort *(Stable, Comparison Sort)*

Bubble Sort is one of the simplest sorting algorithms: it works by repeatedly stepping through the array and comparing two items at a time, swapping them if they are in the wrong order (only respect each other). So, at every step the largest Element is brought at the right place, and the forward part of the array has been arranged a bit more.

The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

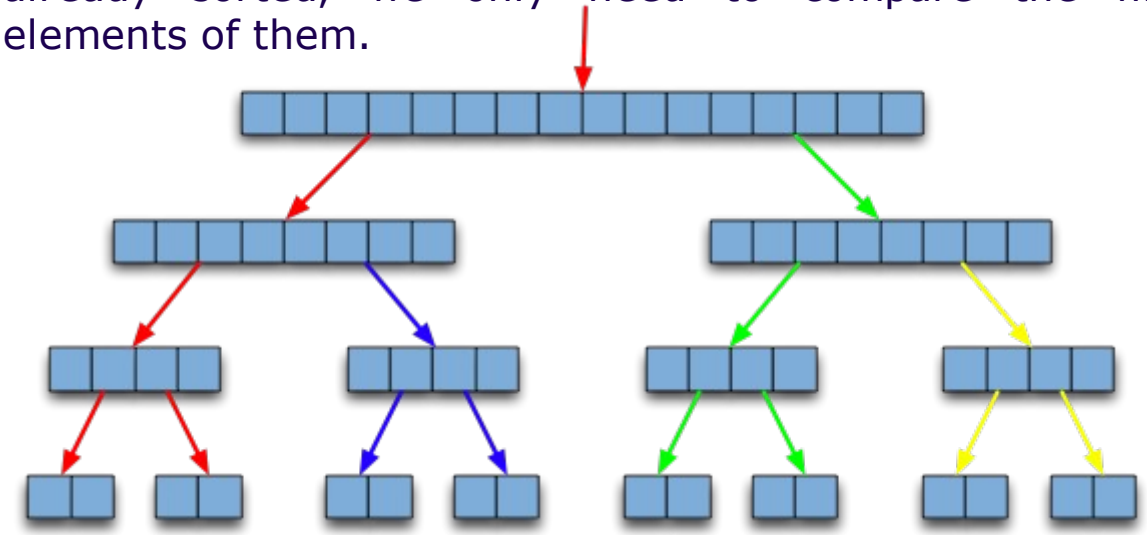


MergeSort *(Stable, Comparison Sort)*

MergeSort is a classical Example of a "*Divide and Conquer*" algorithm, which acts like a "Direct Elimination Football Tournier": the 1st and 2nd Element of the Array are compared, the 3rd and 4th, etc. until the end.

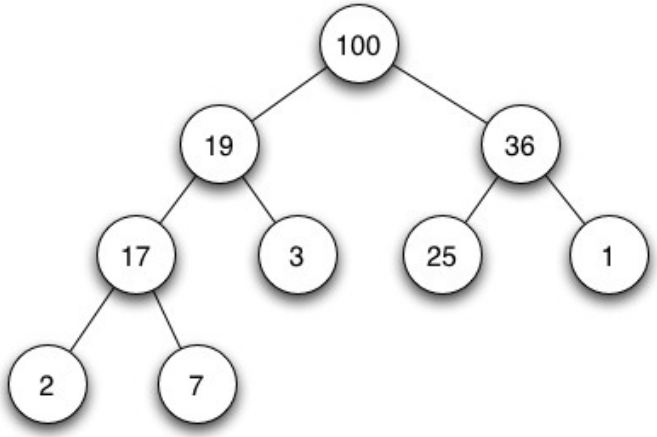
Then this new "blocks" are again compared (1st and 2nd , etc.), until the whole sequence forms a unique block.

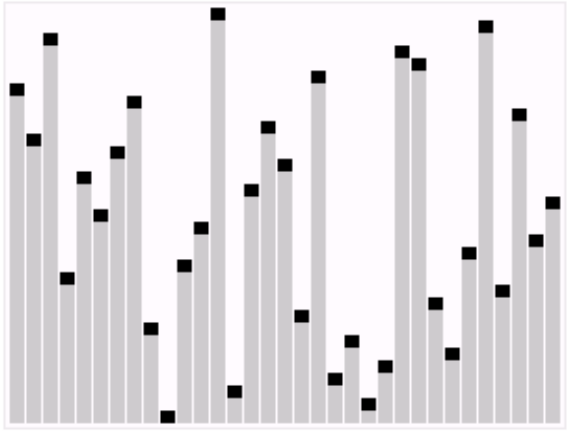
Comparing two blocks is quite easy: since this blocks are already sorted, we only need to compare the first elements of them.



HeapSort *(Not Stable, Comparison Sort)*

Heapsort is a good example of the usage of a particular Data-Structure: it first inserts the input-elements into a *heap* (a special data structure, w.l.o.g. a max-heap). The largest value is iteratively extracted from the heap (operation extract-max of max-heap structure) until nothing remains: the values have been extracted in sorted order, and so the sorted sequence is already built (namely in the inverse sense of the extraction).





Other Sorting Algorithms

These 4 were only some of the most famous ones (together with Quicksort), but there are a lot of other Algorithms for Sorting Purposes: here just a few names:

Exchange Sorts	Quicksort, Bubble sort, Cocktail sort, Comb sort, Gnome sort,
Selection Sorts	Heapsort, Selection sort, Smoothsort, Strand sort
Insertion Sorts	Insertion sort, Shell sort, Tree sort, Library sort, Patience sorting
Merge Sorts	Mergesort
Non-Comparison Sorts	Bucket sort, Radix sort, Counting sort, Pigeonhole sort, Tally sort
Others	Topological sorting, Network sorting

Running Times & Comparison Example

Running Times & Comparison Example

Selectionsort: $O(n^2)$
Bubblesort: $O(n^2)$
Mergesort: $O(n \cdot \log(n))$
Heapsort: $O(n \cdot \log(n))$ in Worst Case
Quicksort: $O(n \cdot \log(n))$ in Average, $\Omega(n^2)$ in Worst Case

Remark:

As already said, in a practical Implementation also other factors are important (Space needed, Data Movement, Data Structure used, Implementation in Machine Code, etc.).

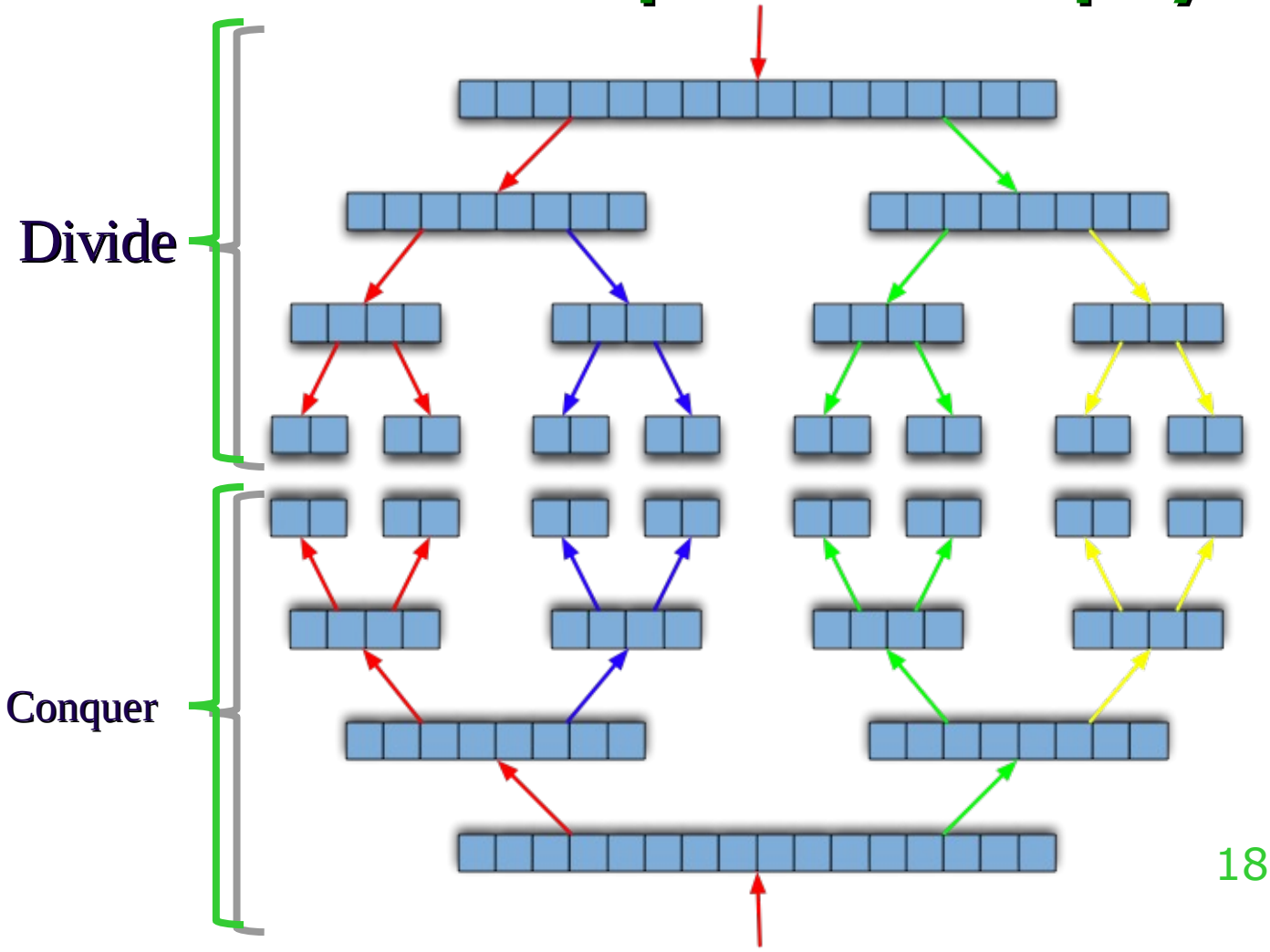
Later: "In-Place" Algorithms…

Divide & Conquer Philosophy

- Introduced in 1960 by Anatolii Karatsuba, the main Idea behind the algorithm design paradigm *"Divide & Conquer"* is to solve a given problem by **dividing it in smaller Subproblems** of the same kind (Divide-Step), solving them separately (again by dividing them, until they are small enough to be directly solved) and, from this sub-solutions, build the global solution for the main Problem (Conquer-Step).

- The fact that the subproblems are independent is a great property, since these tasks are naturally adapted for parallel systems (execution in multiprocessor machines), and obtaining the global solution just by reading the solutions of the single PC's.

Divide & Conquer Philosophy



Divide & Conquer Philosophy

- For this kind of Algorithms, we have a great Tool – The Master Theorem – which helps us to easily compute much Running-Times, since the recurrence formulas are almost always of the same Form (will see later…).

- Some Examples of Algorithms of this Type are:

  - Quicksort
  - Mergesort
  - Binary Search
  - Discrete Fourier-Transform
  - Strassen Matrix-Multiplication
  - Tower of Hanoi Puzzle
  - …

The Main Idea of Quicksort

The Main Idea of Quicksort

- We're given an array: we have to choose a Pivot (how to make this choice is the most discussed part of the algorithm). In the previous example we keep the element of the array in the middle position.
- Then we partition the array by putting to the right of the pivot all elements bigger than it, and to the left the smaller ones.
- The new array has now the pivot (14 in the example) at the correct place, and we have 2 smaller arrays, which we can separately sort by iterating steps a) and b) for each new sub-array.
- We repeat a), b) and c) until it doesn't make any sense more to divide the array in 2 parts (for example if it contains only the pivot or is sufficiently small).

<u>In this way we have ordered all our elements!</u>

We just have to put them back all together (if for example we used more PC's to do the job). *[Conquer Phase]*

<u>Note:</u> The Conquer Phase in Quicksort algorithm is, at least theoretically, quite trivial…

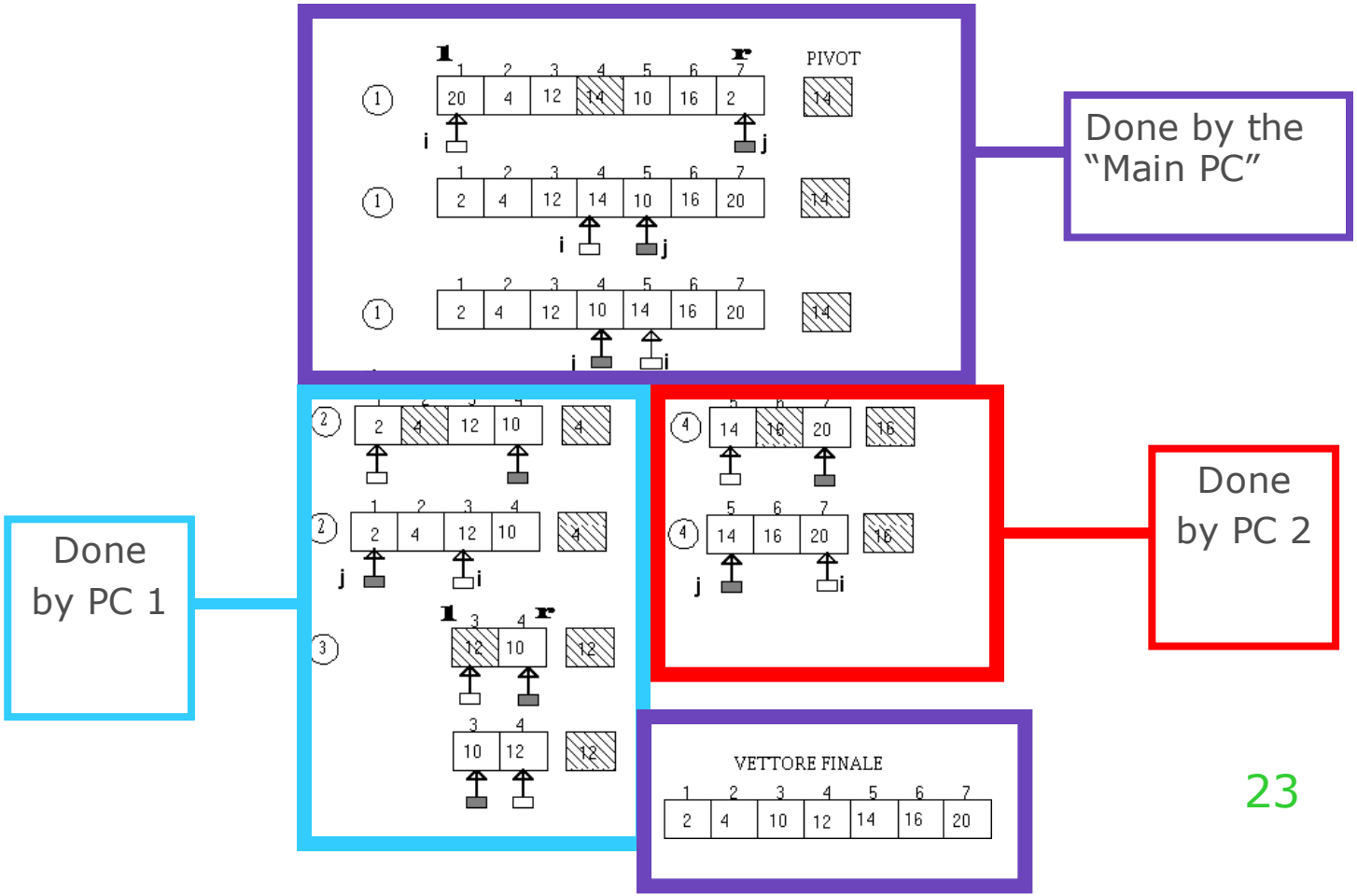
Advanatages and Disadvantages of the Quicksort Algorithm

- More advantages than disadvantages (after all, Quicksort is one of the 10 best algorithms of the XX century!)
- Divide & Conquer Advantages
- Speed (as advantage and disadvantage)
- The Stack Problem
- Advantage of being an "In-Place Algorithm"

Divide & Conquer Advantages

- The divide step allows us to work with more PC's, which don't need to be connected to each other:

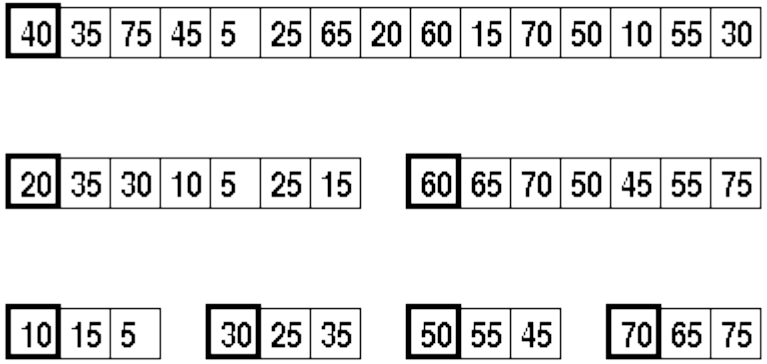
  for example in the previous example we could divide the whole job in the following way:



Running Time

> The Best Possible Case:

The best possible case happens when we choose each time the **median** as the new pivot and, consequentially, we divide every array in 2 parts of the (rounded) same length.



In this case the algorithm runs in $O(n \log n)$

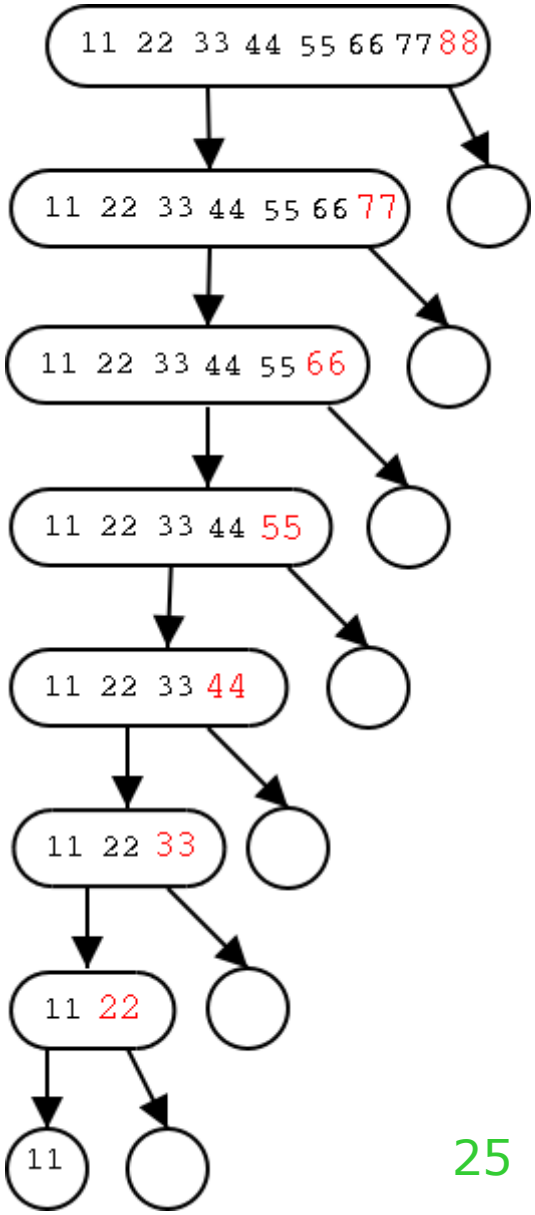
More precisely we need $1{,}39 \cdot n \cdot \log n$ steps.

Running Time

- The Worst Possible Case:

The worst possible case for the algorithm happens when we always choose the biggest (or the smallest) element of the array as the Pivot.

Paradoxically, this is the case of an already sorted array: if we choose the last element (like in the example) or the first one as Pivot, the array is divided in an extremely uneven way!



Running Time

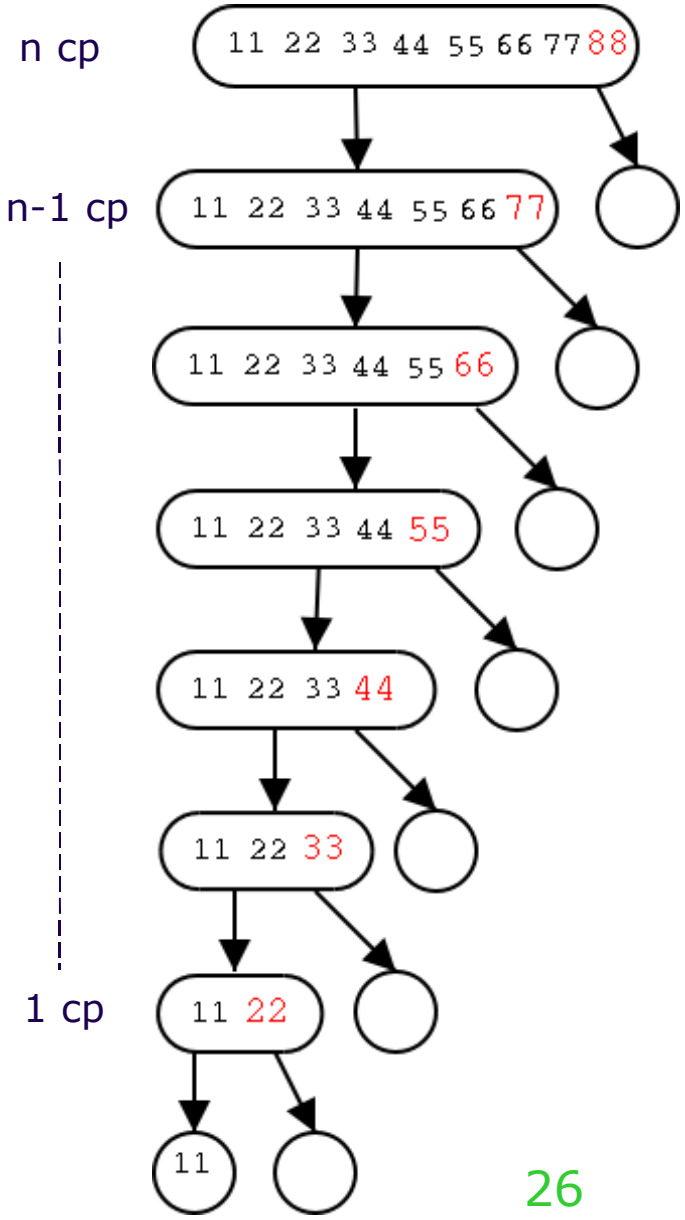
> The Worst Possible Case:

In this way, the total number of comparisons that we do is:

$$\#\text{cp} = \sum_{i=1}^{n} n - i$$

$$= \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$$

$$= O(n^2)$$



Running Time

- Improving the Algorithm:

It is easy to understand that the main problem of quicksort is how we choose the pivots:

We can choose them randomly: this will make the algorithm fast in the best possible case, but will still be very slow in the worst possible case.

A first solution could be the following: by choosing the pivot, choose 3 numbers uniformly at random (in spite of one) and keep the median of the 3 numbers as pivot.

This method will

1) improve the worst possible case
2) make it less probable to happen

But in the best case, we spend too much time to make "useless" comparisons for determining the pivots.

Running Time

- Improving the Algorithm:

  A second possibility is for example to exactly find out the median (for example with the help of another algorithm like quickselect).

  Doing so we will always have the best possible case, but this has a high price: in fact the algorithm will be slower in the best cases, because of the usage of "quickselect".

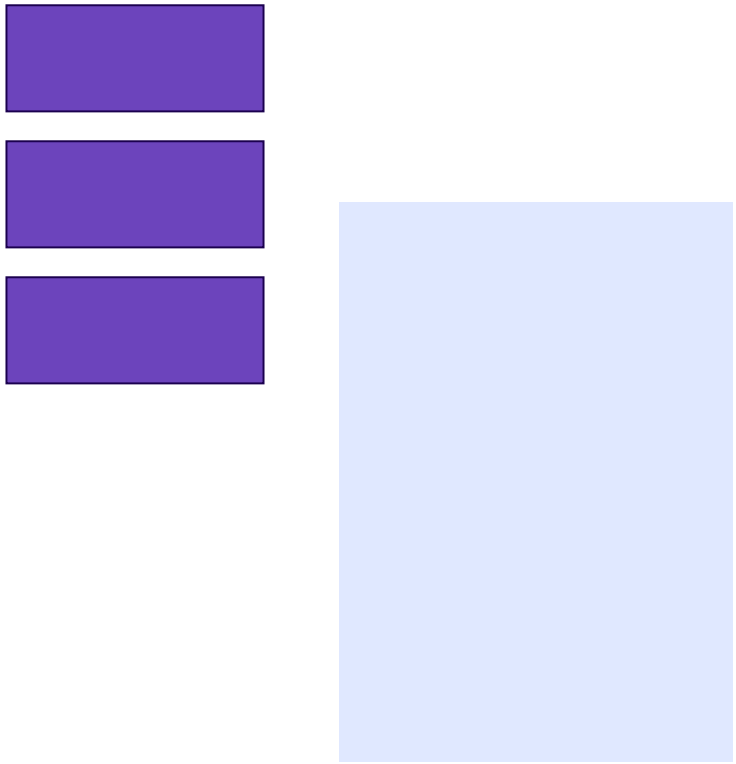
- Conclusion:

  There doesn't exist any "fast" method of choosing the pivot, which improves the algorithm in all possible cases!

The Stack Problem

- Definition:

  In computer science, a stack is an abstract data type and data structure based on the principle of “Last In First Out” (LIFO).

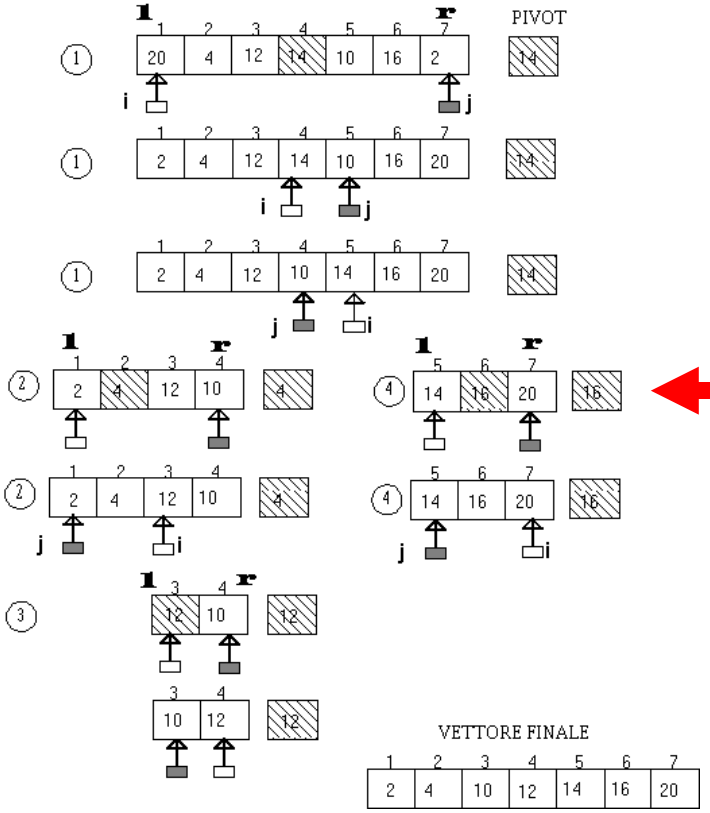


The Stack Problem

- The Quicksort Case:

In our case we put in the stack the set of jobs we still have to do (which arrays we still have to sort).

Example:



Do 2a

Do 3

Do 4

The Stack Problem

- The Quicksort Case:

A problem could consist in having a stack which is not big enough.

In the worst possible case the stack could contain n jobs!

This would be a problem.

To solve this, we can decide to perform first the shortest jobs, and then the longer ones.

So the Quicksort's worst possible case will give us a stack with at most 2 elements, and the worst possible case for the stack will be the best possible case for the Quicksort Algorithm (in this case the stack will have at most

$$\log(n)$$

elements).

"In Place" Algorithms

> Definition:

In computer science, an **in-place algorithm** is an algorithm which transforms a data structure using a small (constant, i.e. *O*(1)) amount of extra storage space.

Quicksort just requires a small extra space for the stack, namely at most:

$$O(\log n)$$

This is usually an advantage: between two identically algorithms (except for the needed extra storage space), the better one is the one which uses less extra space.

Quicksort: a Pseudocode

```
function quicksort(array)
        var list less, equal, greater
        if length(array) ≤ 1
                return array
        Select a pivot value pivot from array
        for each x in array
                if x < pivot then append x to less
                if x = pivot then append x to equal
                if x > pivot then append x to greater
return concatenate(quicksort(less), equal, quicksort(greater))
```

Quicksort: a C++ Implementation

```
void QuickSort(int& array[], int begin, int end)
{
   int pivot, l, r;
   if (end > begin)
   {
      pivot = array[begin];
      l = begin + 1;
      r = end   + 1;
      while(l < r)
         if (array[l] < pivot)
            ++l;
         else
         {
            --r;
            swap(array[l], array[r]);
         }

      --l;
      swap(array[begin], array[l]);
      QuickSort(array, begin, l);
      QuickSort(array, r, end);
   }
}
```

Theorem of Optimality

Every Algorithm that sorts n Elements using "if-statements" (i.e. comparisons) requires at least $O(n \cdot \log n)$ comparisons.

Proof:

[On the Blackboard]

Consequence:

Quicksort is a very good algorithm, because it works (in average) in

$$O(n \cdot \log n)$$

A "CounterExample": BucketSort

(Stable, Non-Comparison Sort)

As one could guess from this theorem, there must also be sorting algorithms which don't need to compare any items: how is it possible?

If we know some information about the Input-Type, then we can avoid any comparison by just using this information.

Let's assume that the elements of the input come from the Universe **U={1, 2, … N}** (or, more generally, a finite & discrete universe that we already know).

Then the Idea is very simple: instead of "sorting" the elements, we just "count" for every jÎU how many times does j appear in the sequence, and we add +1 to our counter of j.

At the End, we simply read the counter states and write as many times Element j as its counter says.

Clearly the running time is O(n) for reading the sequence and O(N) to initialize the Counter-Array (Buckets), so the final running time is a **linear O(n+N).**

A "CounterExample": BucketSort

The Main Difference: Randomness

As already mentioned in the "Deterministic Quicksort" section, the choice of the Pivot is crucial for a good partitioning, and bad choices can cause the algorithm to partition the input unevenly and make a lot of comparisons which in fact could have been avoided.

> Definition: Randomized Algorithm

Here we consider a **Randomized Algorithm** (or Probabilistic Algorithm), i.e. an algorithm which employs a degree of randomness as part of its logic (assuming access to a pseudorandom number generator).

The aim is to use the random bits as an auxiliary input to guide the behavior, in the hope of achieving good performance in the "average case".

The Main Difference: Randomness

'Randomized Quicksort' is the simplest way of setting 'Deterministic Quicksort' into a Randomized Algorithm:

We always choose the Pivot u.a.r. *(uniformly at random)* among the keys!

Remark:

An equivalent way of doing this would be to take the input array and randomly shuffle it, then procede as deterministic quicksort (i.e. choose the first key as the Pivot).

The Main Difference: Randomness

It's important to note the difference between deterministic- and randomized-quicksort:

the former assumes that the input is given in random order, hence there can be (very) bad inputs (an almost sorted sequence).

In contrast, for the randomized version there's nothing like a "bad" input, since the randomness is introduced internally by the algorithm without having to rely on a "nice" input distribution.

Analysis: Solving the Recursion

In the following we would like to compute exactly the expected number of comparison (between the elements) made by Randomized Quicksort.

We assume that the elements **are all different** (the case were two are the same is avoided just for the explanation to be easier).

W.l.o.g. we assume that the input is a random permutation of the universe **S = {1, 2, … , n}**.

Def: The Rank of the element x in S is:

$$rk(x) := 1 + |\{y \in S \mid y < x\}|$$

(i.e.: *x is the rk(x)-smallest Element in S*)

Analysis: Solving the Recursion

- **Remark 1:** ***(Uniform Distribution)***

If we pick an element X u.a.r from {1, … , n}, then

$$\Pr[X = j] = \frac{1}{n}, \forall j \in \{1,\ldots,n\}$$

- **Remark 2:** ***(Law of Total Probability)***

Let X,Y be two random Variables on a Common Probability Space ({1, … , n}, Pr[.])

Then:

$$E[X] = \sum_{i=1}^{n} E[X \mid Y = i] \cdot \Pr[Y = i]$$

- **Def:**

Let X_n denote the expected number of comparisons made when sorting a set of n keys.

($X_0 = X_1 = 0$

Analysis: Solving the Recursion

- Noting that we need |S|-1 comparisons to split the array and then the two subproblems are totally independent, we get:

$$E[X_n] = \sum_{i=1}^{n} \underbrace{E[X_n \mid rk(pivot) = i]}_{(i-1)+E[X_{i-1}]+(n-i)+E[X_{n-i}]} \cdot \underbrace{\Pr[rk(pivot) = i]}_{1/n}$$

$$= n - 1 + \frac{1}{n} \cdot 2 \cdot \sum_{i=1}^{n} E[X_{i-1}]$$

- The Recursion becomes also:

$$E[X_n] =: x_n = \begin{cases} 0 & \text{, if } n=0 \\ n - 1 + \frac{2}{n} \cdot \sum_{i=1}^{n} x_{i-1} & \text{, otherwise} \end{cases}$$

Analysis: Solving the Recursion

$$E[X_n] =: x_n = \begin{cases} 0 \\ n-1+\frac{2}{n}\cdot\sum_{i=1}^{n} x_{i-1} \end{cases}$$

- For n≥2 we get:

$$nx_n = n(n-1)+2(x_0+x_1+\ldots+x_{n-2}+x_{n-1})$$

$$(n-1)x_{n-1} = (n-1)(n-2)+2(x_0+x_1+\ldots+x_{n-2})$$

and by subtraction:

$$nx_n-(n-1)x_{n-1} = 2(n-1)+2x_{n-1}$$

$$\Updownarrow$$

$$nx_n = 2(n-1)+(n+1)x_{n-1}$$

$$\Updownarrow$$

$$\underbrace{\frac{x_n}{2(n+1)}}_{=:f_n} = \frac{(n-1)}{n(n+1)} + \underbrace{\frac{x_{n-1}}{2n}}_{=f_{n-1}}$$

Analysis: Solving the Recursion

- Hence we got:

$$\begin{cases} f_n := \dfrac{x_n}{2(n+1)} \\ f_n = \dfrac{1}{(n+1)} - \dfrac{1}{n(n+1)} + f_{n-1}, n \geq 2 \end{cases}$$

Therefore:

$$f_n = \underbrace{\left(\frac{1}{(n+1)} + \frac{1}{n} + \ldots + \frac{1}{3}\right)}_{= H_{n+1} - \frac{3}{2}} - \underbrace{\left(\frac{1}{n(n+1)} + \frac{1}{(n-1)n} + \ldots + \frac{1}{2 \cdot 3}\right)}_{\sum_{i=2}^{n} \frac{1}{i(i+1)} = \sum_{i=2}^{n} \left(\frac{1}{i} - \frac{1}{i+1}\right) = \frac{1}{2} - \frac{1}{n+1}} + \underbrace{f_1}_{=0}$$

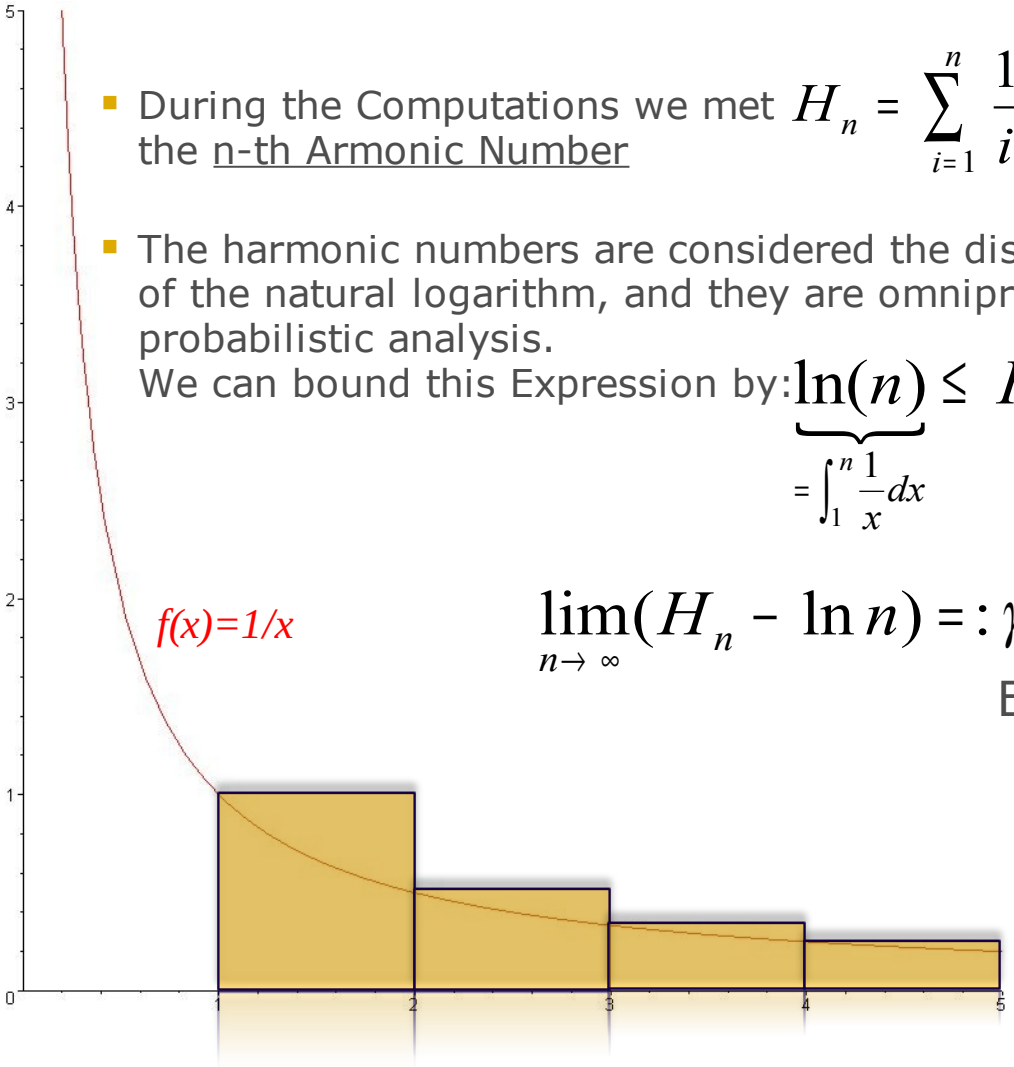
$$= H_{n+1} - 2 + \frac{1}{n+1} = H_n - 2 + \frac{2}{n+1}$$

- Finally:

$$\begin{aligned} x_n &= 2(n+1)H_n - 4n \\ &= O(n \cdot \log n), n \geq 0 \end{aligned}$$

Analysis: Solving the Recursion

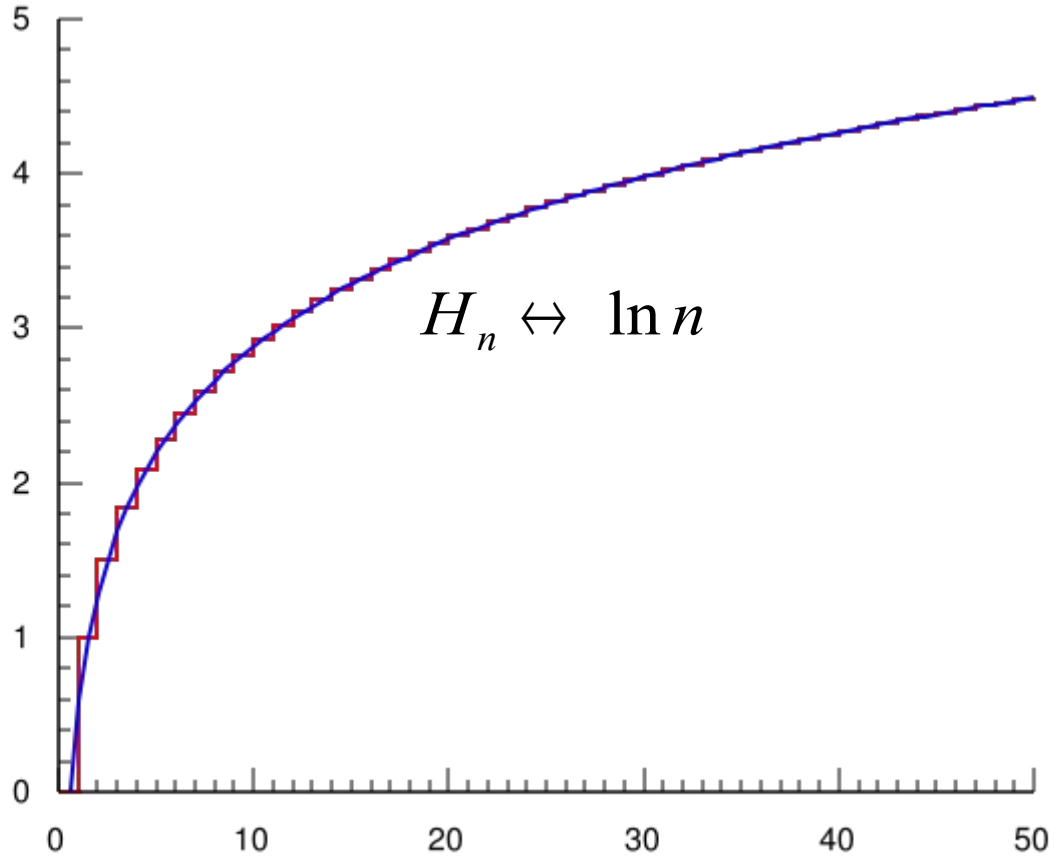
- During the Computations we met $H_n = \sum_{i=1}^{n} \frac{1}{i}$ the n-th Armonic Number
- The harmonic numbers are considered the discrete analogue of the natural logarithm, and they are omnipresent in discrete probabilistic analysis.
  We can bound this Expression by: $\underbrace{\ln(n)}_{=\int_1^n \frac{1}{x}dx} \leq H_n \leq \ln(n) + 1$



$$\lim_{n \to \infty}(H_n - \ln n) =: \gamma = 0.57721...$$

Euler Constant

Analysis: Solving the Recursion

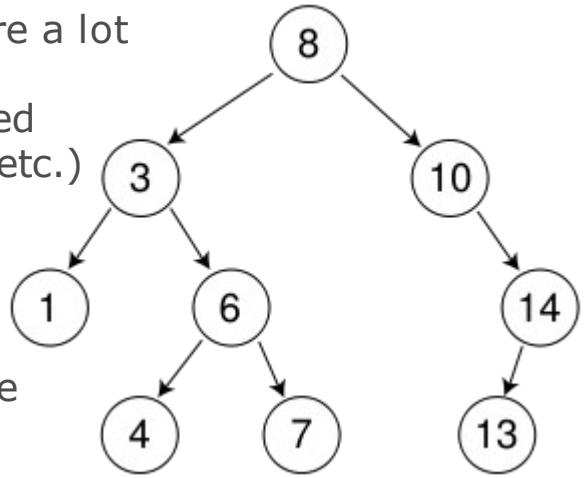


Analogy with Random Search Trees

- As maybe someone as already noticed, the development of the Recursion of randomized Quicksort is exactly a random search tree, i.e. a data structure where the elements are stored s.t. one is called the **root** and every node has at most two children (left = smaller, right = bigger). Every "subroot" **is chosen randomly**, and the sets below partitioned in smaller/bigger than this subroot iteratively.

- In such a data structure there are a lot of properties which are easy to derive (expected height, expected height of the smallest element, etc.)

  What we have computed with the analysis of Quicksort corresponds exactly to the **Expected Overall Depth** of the "Quicksort-Tree".



The Master Theorem

> The Recursion

As promised during the "Divide & Conquer" section, we have a great tool for solving the Recursions of this kind of Algorithms.

In fact the strategy consists in dividing the main problem (of size n) in a subproblems of size n/b.

The running-time recursion is thus:

$$T(n) = \alpha \cdot T(n/\beta) + f(n)$$

Where f(n) is the time needed for splitting the problem in the smaller subproblems (+ ev. The time needed to merge the solutions in the "Conquer-Phase").

The Master Theorem

- Master Theorem

Let $a \geq 1$, $b > 1$ and C ≥0 constants, $c_1(n),\ldots,c_\alpha(n) \leq C$ functions and f(n) a positive function (n integer).

If T(n) is a function with T(1)=0, which satisfies the recursion

$$T(n) = T(n/\beta + c_1(n)) + \ldots + T(n/\beta + c_\alpha(n)) + f(n)$$

then the solution has the following behavior:

$$T(n) = \begin{cases} \Theta(n^{\log_\beta \alpha}) & \text{, if } f(n) = O(n^{\log_\beta \alpha - \varepsilon}) \quad \text{for an } \varepsilon > 0 \\ \Theta(f(n)\cdot \log n) & \text{, if } f(n) = \Theta(n^{\log_\beta \alpha}(\log n)^\delta) \quad \text{for a } \delta \geq 0 \\ \Theta(f(n)) & \text{, if } f(n) = \Omega(n^{\log_\beta \alpha + \varepsilon}) \quad \text{for an } \varepsilon > 0 \end{cases}$$

The Master Theorem

$$T(n) = T(n/\beta + c_1(n)) + \ldots + T(n/\beta + c_\alpha(n)) + f(n)$$

$$T(n) = \begin{cases} \Theta(n^{\log_\beta \alpha}) & \text{, if } f(n) = O(n^{\log_\beta \alpha - \varepsilon}) \quad \text{for an } \varepsilon > 0 \\ \Theta(f(n) \cdot \log n) & \text{, if } f(n) = \Theta(n^{\log_\beta \alpha} (\log n)^{\delta}) \quad \text{for a } \delta \geq 0 \\ \Theta(f(n)) & \text{, if } f(n) = \Omega(n^{\log_\beta \alpha + \varepsilon}) \quad \text{for an } \varepsilon > 0 \end{cases}$$

- Application to Quicksort

  In Expectation, the random choice of every Pivot divides every sequence in equal size, so the recursion is

  $$T(n) = T(n/2) + T(n/2) + (n-1)$$

  We can also read:

  $$\alpha = 2 = \beta \text{ and } f(n) = \Theta(n^{\log_2 2}(\log n)^0) = \Theta(n)$$

  $$\Downarrow$$

  $$T(n) = \Theta(n \cdot \log n)$$

The Main Idea of Quickselect

> Problem

Given a Set S of keys, suppose we want to know the Middle Element (= Median) or, more generally, the k-smallest element in S (i.e. the element of rank k in S).

> Idea

The first Idea would be to simply sort the array and output the k-th element of it: that works in expected O(n log(n)), but this is inadmissible: after all for outputting the smallest Element we would need just O(n) comparisons…

In fact a moment of reflection shows that if all we care about is the element of rank k, in each call of quicksort() one of the two sub-calls is irrelevant (eventually both, if the pivot is our element of rank k).

The Main Idea of Quickselect

- The Procedure

function quickselect(k,S)

x := u.a.r. from S;
Split S into $S^{<x}$, x, $S^{>x}$;
r:= $|S^{<x}|+1$; *(i.e. $r=rk_S(x)$)*

If k<r return quickselect(k, $S^{<x}$);
Elseif k=r return x;
Else *(i.e. k>r)* return quickselect(k-r, $S^{>x}$);

Analysis of Quickselect

> Analysis

Let's define t(k,n) as the expected number of comparisons among elements in S calling quickselect(k,|S|=n).

t(1,1)=0 and for n≥2:

$$t(k,n) = \underbrace{n-1}_{Split-Step} + \sum_{l=1}^{n} \underbrace{\Pr[Pivot = l]}_{1/n} \cdot E[t(k,n) \mid Pivot = l]$$

$$= n - 1 + \frac{1}{n} \cdot \left(\underbrace{\sum_{l=1}^{k-1} t(k-l, n-l)}_{quickselect(k-l, S^{>x})} + \underbrace{\sum_{l=k+1}^{n} t(k, l-1)}_{quickselect(k, S^{<x})} \right)$$

Analysis of Quickselect

Solving this recurrence looks hopeless, but we are just interested in the asymptotics, so we omit the parameter k, setting:

$$t_n := \max_{k=1..n} t(k,n)$$

And assuming for the recursive call always the size of the larger of the two sets $S^{<x}$, $S^{>x}$. That is, $t_1=0$ and for $n \geq 2$:

$$t_n \leq n - 1 + \frac{1}{n} \cdot \sum_{l=1}^{n} t_{\max\{l-1,n-l\}}$$

CLAIM: $t_n \leq cn$ for some real constant c>0.

Analysis of Quickselect

Proof:

$t_1=0$ is a safe basis for induction.

For $n \geq 2$ we must show:

$$n-1+\frac{1}{n}\cdot\sum_{l=1}^{n}\underbrace{t_{\max\{l-1,n-l\}}}_{Induction:\leq c\cdot\max(l-1,n-l)} \overset{?}{\leq} cn$$

$$\Updownarrow$$

$$\underbrace{\sum_{l=1}^{n}\max\{l-1,n-l\}}_{=\begin{cases}\frac{3}{4}n^2-\frac{1}{2}n & \text{, if n even}\\ \frac{3}{4}n^2-\frac{1}{2}n-\frac{1}{4} & \text{, if n odd}\end{cases}} \overset{?}{\leq} \frac{c-1}{c}n^2+\frac{1}{c}n$$

Holds with c=4.

Analysis of Quickselect

Theorem:

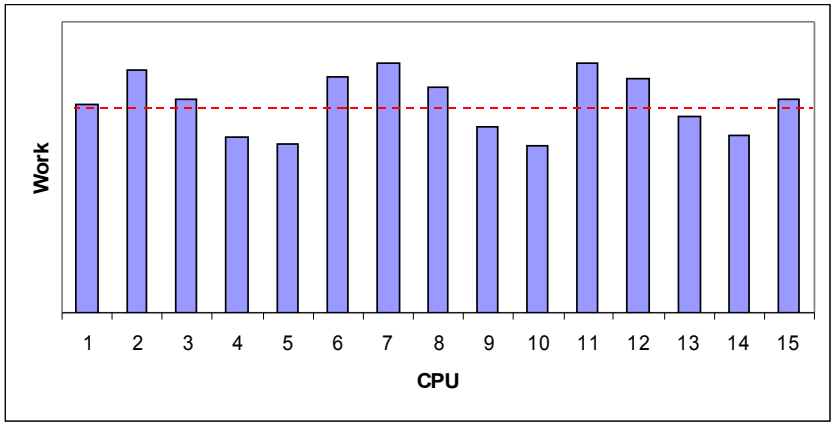
The expected number of comparisons (among input numbers)performed by a call quickselect(k,|S|=n) is **at most 4n = *O*(n)**.

Parallel Implementation

- The main idea is to divide the job between **p processors**, so that every of them can work independently from the others.

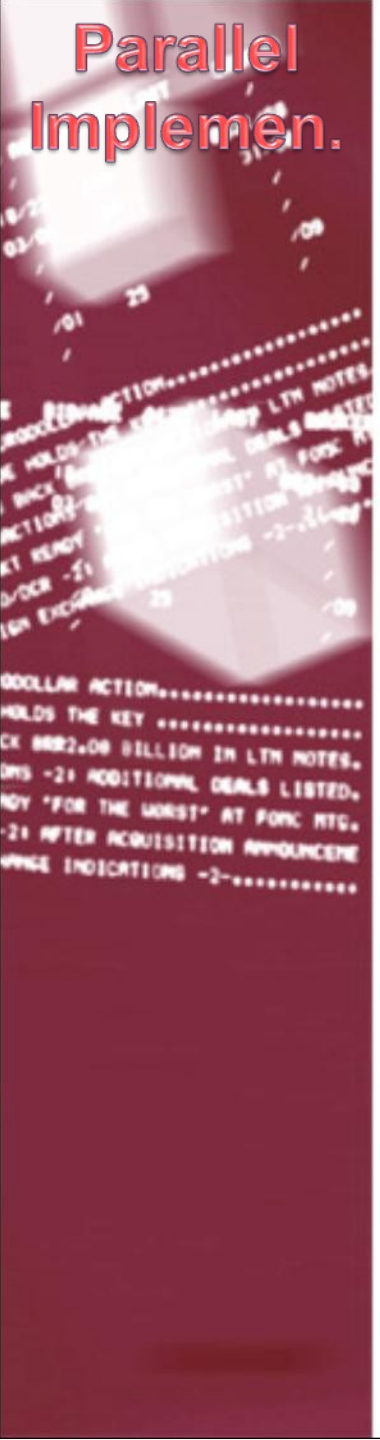
Note: A good job division will respect this criteria:

$$W_i \approx \frac{W_{TOT}}{n}$$



W_i := Work done by CPU i

W_{TOT} := Total work



Parallel Implementation

The main question is also: how do we divide the job between the p processors?

Assumptions:

- We work with p identical CPU's
- The length of the unsorted array is n=p·q
  (with q quiet big)

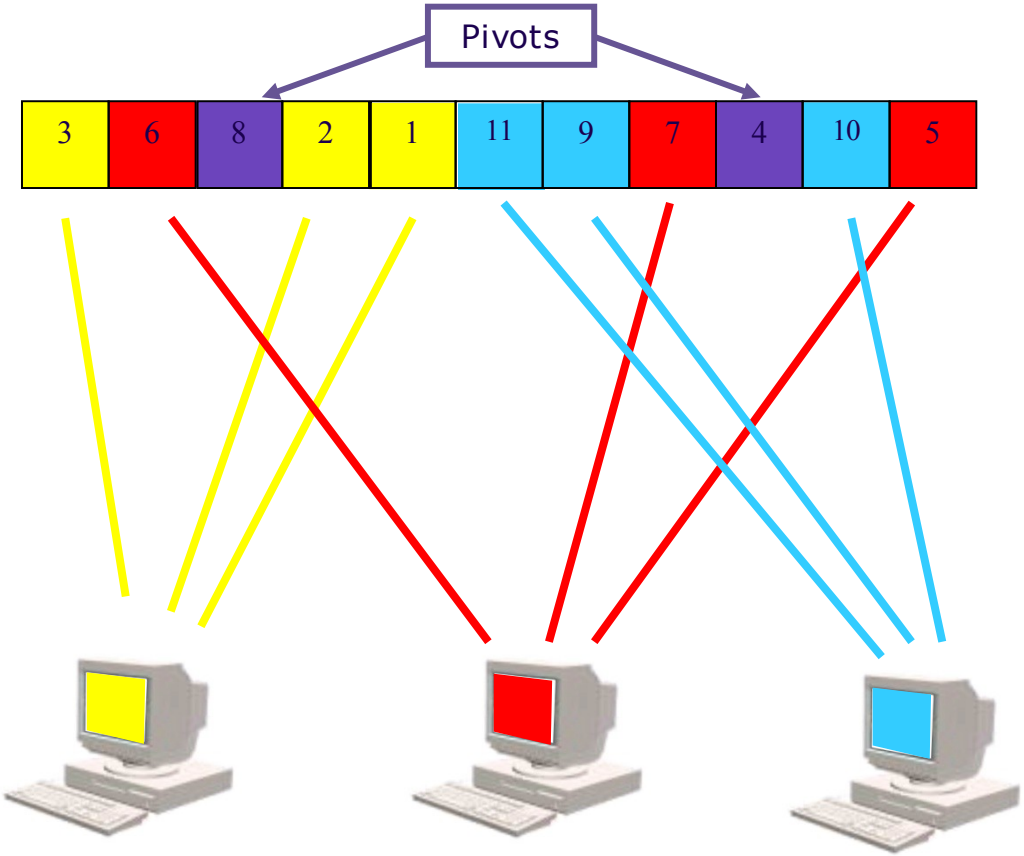
We choose $p-1$ pivots s_i , s.t. $s_i < s_{i+1} \; \forall \, i$

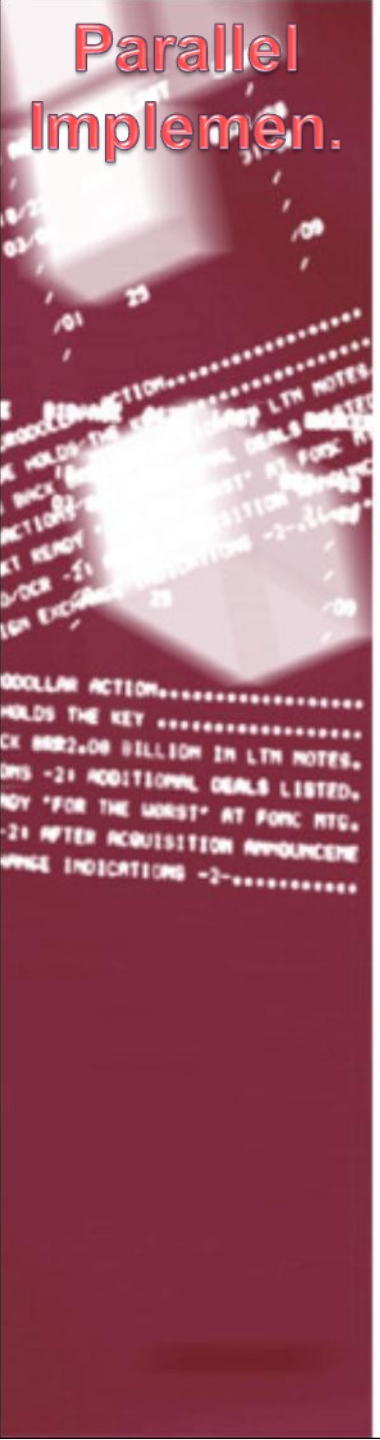
The i-th CPU chooses, from the unsorted vector, all the components in:

$[s_{i-1}; s_i]$, where $s_0 := -\infty$, $s_p := +\infty$

Parallel Implementation

The best choice for the pivots is the one which divides the array in p equal parts.





Parallel Implementation

> How to choose the p-1 pivots?

A good way to choose the pivots is the following: every processor chooses m pivots at random and sends these pivots to the first processor.

Then the first processor sorts these pivots (using quicksort) and chooses the m·i-th+m/2 elements of this new array.

These numbers will be used as pivots.

Literature

- Angelika Steger - ***Algorithmen und Komplexität***, Skript zur Vorlesung 2006/2007
- Angelika Steger, Emo Welzl, Peter Widmayer – ***Algorithms, Probability and Computing***, Skript for Semester 2007
- http://www.wikipedia.org
- http://www.sci.csuhayward.edu/~billard/cs3240/node32.html
- http://www.sikh-history.com/education/computers/sort/index.html
- http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/quick/quicken.htm